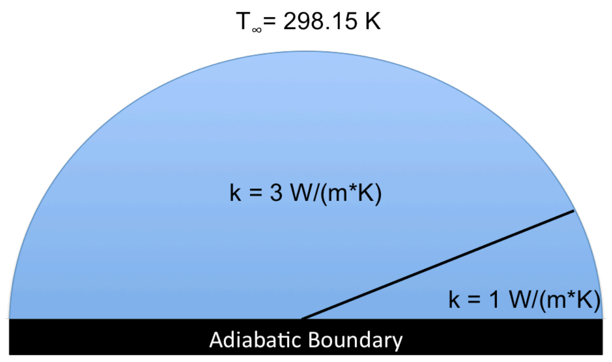
Conduction Heat Transfer Project Report

By Mikhail Lara

Description of Geometry and Problem Formulation

The problem of interest is transient conduction in a long cylinder of heterogeneous thermal conductivity that is exposed to a convective environment characterized by a constant far-field temperature and a constant convection coefficient. This problem is studied using both an in-house finite-volume-based numerical solver and the commercial software COMSOL. The in-house numerical solver was written using JAVA and compared against the results generated by COMSOL simulations to compare the accuracy of the implemented finite-volume approach to the conventional finite-element technique. The geometry specified by the project description is a circular cylinder with a diameter of 0.04m and diameter-to-length ratio of 0.26. The material properties of the cylinder are specified such that the entire thermal conductivity of the cylinder is 3 W/(m*K) except for a 45° region where it is set to 1W/(m*K). In addition to these material properties, the entire geometry is assigned a volumetric generation that is spatially uniform for the entire duration of the simulation.



$T(t=0) = 298.15$ K

$h = \{2\ W/(m^2*K),\ 10\ W/(m^2*K)\}$

$q'''(t) = C_1 t^6 + C_2 t^5 + C_3 t^4 + C_4 t^3 + C_5 t^2 + C_6 t + C_7$

C_1=-1.7031E-11	C_2=1.3146E-07	C_3=-1.8011E-04
C_4=1.024E-01	C_5=-2.8133E01	C_6= 3.6444E03
C_7 =-2.7545E03		

Figure 1: Schematic of Simulated Geometry

Strictly speaking, a circular cylinder of this dimension should be studied using a full three-dimensional(3D) computational mesh because it is not sufficiently long enough to guarantee that end effects would not play a major role in the solution of the temperature field. The commonly accepted criteria used to justify reduction of a 3D cylindrical geometry to an equivalent two-dimensional(2D) geometry are a diameter-to-length ratio less than 0.1, symmetry of the material properties, symmetry of the volumetric sources, and symmetry of the imposed boundary conditions. If all these criteria are satisfied, the solution of the full 3D simulation will generate a temperature field solution that exhibits strong 2D symmetry. Since three of these four criteria are satisfied, the simulations performed in this study were conducted on a 2D geometry of a circle with a diameter of 0.04m. Although the reduction to a 2D geometry was performed in this study, it is important to note that this assumption is likely a source of discrepancy between the results presented here and the results of a full 3D simulation.

The simplified 2D geometry is reduced further by taking a line of symmetry that cuts half way through the 45° region and splits the circular domain into two symmetrical hemispheres. Taking advantage of the system symmetry is useful with regards to spatial resolution because it allows the simulations to produce twice the spatial resolution using a constant number of computational cells. The simplified domain used in the in-house code simulations is a hemisphere with a thermal conductivity of 3 W/(m*K) assigned to 157.5° region and a thermal conductivity of 1 W/(m*K) assigned to the remaining 22.5° region. To guarantee that the results of simulations on this geometry are representative of the circular geometry, adiabatic boundary conditions are designated along the lines located at angular positions of 0° and 180° in addition to the convection boundary conditions designated along the remaining domain boundary. A schematic of the simplified computational domain with the relevant initial conditions and boundary conditions is illustrated in Figure 1.

Numerical Scheme and Solution Method

The solution method used for the transient in-house numerical solver is based on the finite-volume approach discussed in class. This method is distinct from the standard finite difference technique used for thermal conduction because it does not require the derivation of differencing equations directly from the heat diffusion equation. Instead, this methodology derives the differencing equations by considering energy conservation of control volumes. Adjacent computational cells are linked through the heat fluxes, which is expressed as a first-order approximation of the Fourier heat flux relation, shown in Equation (1), where k_{AB} is the relevant thermal conductivity. The subscripts indicate that the derivatives are evaluated by considering energy flow from Cell A to Cell B.

$$q''_{AB} = \frac{k_{AB}(T_A - T_B)}{\Delta L}$$
(1)

Three different types of control volumes had to be considered in order to derive the differencing equations used for the in-house numerical simulations. These different types of control volumes correspond to the cells located at the maximum radial position(boundary cells), the cell located at a radial position of zero(center cell), and the interior cells located between the center cell and the boundary cells. Differencing equations had to be formulated for each of these because the number and form of the heat fluxes are different depending on the cell type. In addition, the differencing equation for the center cell is distinct from the boundary and interior cells. This is a direct result of the fact that in order to truly close the cylindrical geometry, all the radial mesh lines must intersect to a point located at the center of the hemisphere. This mesh structure is in fact unstructured and is a major source of computational error.

The generic energy balances for the internal cells, boundary cells, and center cells used to derive the differencing equations are indicated by Equations (2)-(4) respectively. The summation term in the differencing equation indicated that for the central cell, which is an n-sided polygon, heat fluxes are considered from all cells that share a boundary with the central cell.

$$\rho C_P \frac{\partial T}{\partial t} r\Delta r\Delta\theta = q''' r\Delta r\Delta\theta + \left[q''_{EP}\Delta r - q''_{PW}\Delta r\right] + \left[q''_{SP} r\Delta\theta - q''_{PN}(r+\Delta r)\Delta\theta\right] \qquad (2)$$

$$\rho C_P \frac{\partial T}{\partial t} r\Delta r\Delta\theta = q''' r\Delta r\Delta\theta + \left[q''_{EP}\Delta r - q''_{PW}\Delta r\right] + \left[q''_{SP} r\Delta\theta - q''_{boundary}(r+\Delta r)\Delta\theta\right] \qquad (3)$$

$$\rho C_P \frac{\partial T}{\partial t}\frac{\pi(\Delta r)^2}{2} = q'''\frac{\pi(\Delta r)^2}{2} + \sum_{j=0}^{m} q''_{jP}\Delta r\Delta\theta$$
(4)

The corresponding differencing equations that were derived by substituting Equation (1) into the energy balances are shown in Equations (5)-(7). It should be noted that since the thermal conductivity in the computational domain is heterogeneous, the nodes addressed in the differencing equations are located at the geometric center of the relevant control volumes.

$$T_P^{i+1}\left[\frac{\rho C_P}{\Delta t} + \frac{k_{EP}}{r(r+0.5\Delta r)(\Delta\theta)^2} + \frac{k_{PW}}{r(r+0.5\Delta r)(\Delta\theta)^2} + \frac{k_{SP}}{(\Delta r)^2} + \frac{k_{PN}(r+\Delta r)}{r(\Delta r)^2}\right] + T_E^{i+1}\left[-\frac{k_{EP}}{r(r+0.5\Delta r)(\Delta\theta)^2}\right] + T_N^{i+1}\left[-\frac{k_{PN}(r+\Delta r)}{r(\Delta r)^2}\right] + T_W^{i+1}\left[-\frac{k_{PW}}{r(r+0.5\Delta r)(\Delta\theta)^2}\right] + T_S^{i+1}\left[-\frac{k_{SP}}{(\Delta r)^2}\right] = T_P^i\left[\frac{\rho C_P}{\Delta t}\right] + q'''_{i+1}$$
(5)

$$T_P^{i+1}\left[\frac{\rho C_P}{\Delta t} + \frac{k_{EP}}{r(r+0.5\Delta r)(\Delta\theta)^2} + \frac{k_{PW}}{r(r+0.5\Delta r)(\Delta\theta)^2} + \frac{k_{SP}}{(\Delta r)^2} + \frac{h(r+\Delta r)}{r\Delta r}\right] + T_E^{i+1}\left[-\frac{k_{EP}}{r(r+0.5\Delta r)(\Delta\theta)^2}\right] + T_W^{i+1}\left[-\frac{k_{PW}}{r(r+0.5\Delta r)(\Delta\theta)^2}\right] + T_S^{i+1}\left[-\frac{k_{SP}}{(\Delta r)^2}\right] = T_P^i\left[\frac{\rho C_P}{\Delta t}\right] + q'''_{i+1} + \frac{hT_\infty(r+\Delta r)}{r\Delta r}$$
(6)

$$T_P^{i+1}\left[\frac{\pi\rho C_P}{2\Delta t} + \frac{\Delta\theta}{\Delta r}\sum_{j=0}^{m}\frac{k_{jP}}{\Delta r}\right] + \frac{\Delta\theta}{\Delta r}\sum_{j=0}^{m}\frac{k_{jP}T_P^{i+1}}{\Delta r} = T_P^i\left[\frac{\pi\rho C_P}{2\Delta t}\right] + \frac{\pi}{2}q'''_{i+1}$$
(7)

$$k_{AB} = \frac{2k_A k_B}{k_A + k_B}$$
(8)

This node placement allows for the use of the harmonic mean, Equation (8), in determining a relevant thermal conductivity for the heat fluxes. Since the problem of interest is transient in nature, solutions for the temperature field had to be solved at various times along the simulated time period of 630s. To guarantee stability in the time stepping approach, the solutions for each time step were solved using a fully implicit formulation of the differencing equations.

These differencing equations in conjunction with the boundary conditions and volumetric generation term constitute a set of linear algebraic equation of the form Ax=b. It can be shown that the system of linear equations is highly diagonally dominant, thereby satisfying the Scarborough Stability Criterion. It is for this reason that the iterative Gauss-Seidel Method was used to solve the system of linear equations.

Results and Discussion

As mentioned, all the in-house simulations were performed using a numerical solver coded in the JAVA programming language. Post-processing of all temperature field data was performed in Tecplot using the built-in contour plotter. It should be noted that before Tecplot can generate a contour plot of the temperature field data, which is imported as a list of unstructured coordinates and temperature values, a triangular mesh must be generated with the built-in triangulation function. The triangulation function creates mesh lines that run through the nodes associated with the centers of the control volumes used in the formulation of the differencing equations. Since the generated mesh lines run through the cell centers, the following plots look like full semicircles were not simulated. This is merely an artifact of how Tecplot triangulates the nodes. A full semicircle spanning 180° was simulated.

The transient temperature fields calculated by the in-house numerical solver for a convection coefficient of 2 W/(m^2*K) are illustrated at 4 different time steps in Figure 2. The simulated temperature distributions indicate for the entire simulated duration, the maximum temperature is located near the center of the hemispherical geometry. This is expected because energy generated at the center of the geometry is the farthest from the convective environment experienced by the boundaries. This significant thermal resistance causes the accumulation of energy near the center, which results in increased temperature. For the first 75% of the simulation duration, the temperature profile demonstrates angular symmetry indicating equal diffusion of energy in all directions. This trend does not persist for the entire duration of the simulation as is apparent at Time = 630s. In the final time step, the effect of the decreased thermal conductivity becomes apparent, as the region of high temperature does not protrude significantly into the portion of the hemisphere associated with lower thermal conductivity. This result is expected because a lower thermal conductivity is associated with increased thermal resistance. The maximum temperature calculated for the entire simulation is of 363 K.

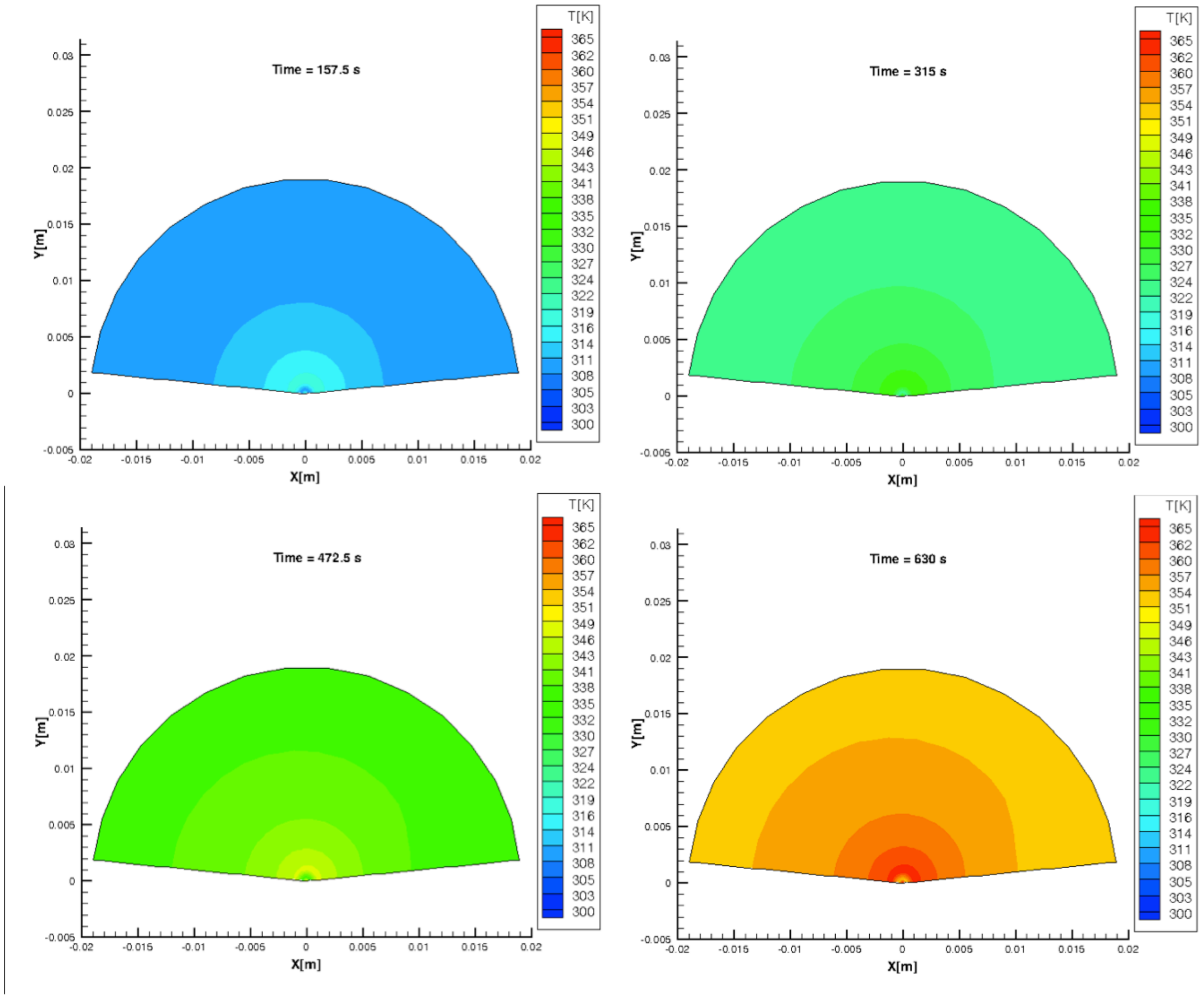


Figure 2: Transient Temperature Disributions for h = 2 W/(m^2*K)

Characteristics of the temperature fields calculated for a convection coefficient of 2 W/(m^2*K) are present in the simulations for a convection coefficient of 10 W/(m^2*K), as shown in Figure 3. For the majority of the simulated time, the temperature fields demonstrate angular symmetry with the temperature increasing to the maximum value as the radial position decreases. The maximum value calculated during these simulations is of 359 K. This value is lower than that found for the simulation with a higher convection coefficient, which is physically accurate because a greater convection coefficient indicates a greater capacity for heat removal. The only major difference between the two simulations is the shape of the contours in the portion of the domain with lower thermal conductivity. In the simulation with the greater convection coefficient, the overall symmetry of the temperature distribution is affected less significantly the when the convection coefficient is lower.

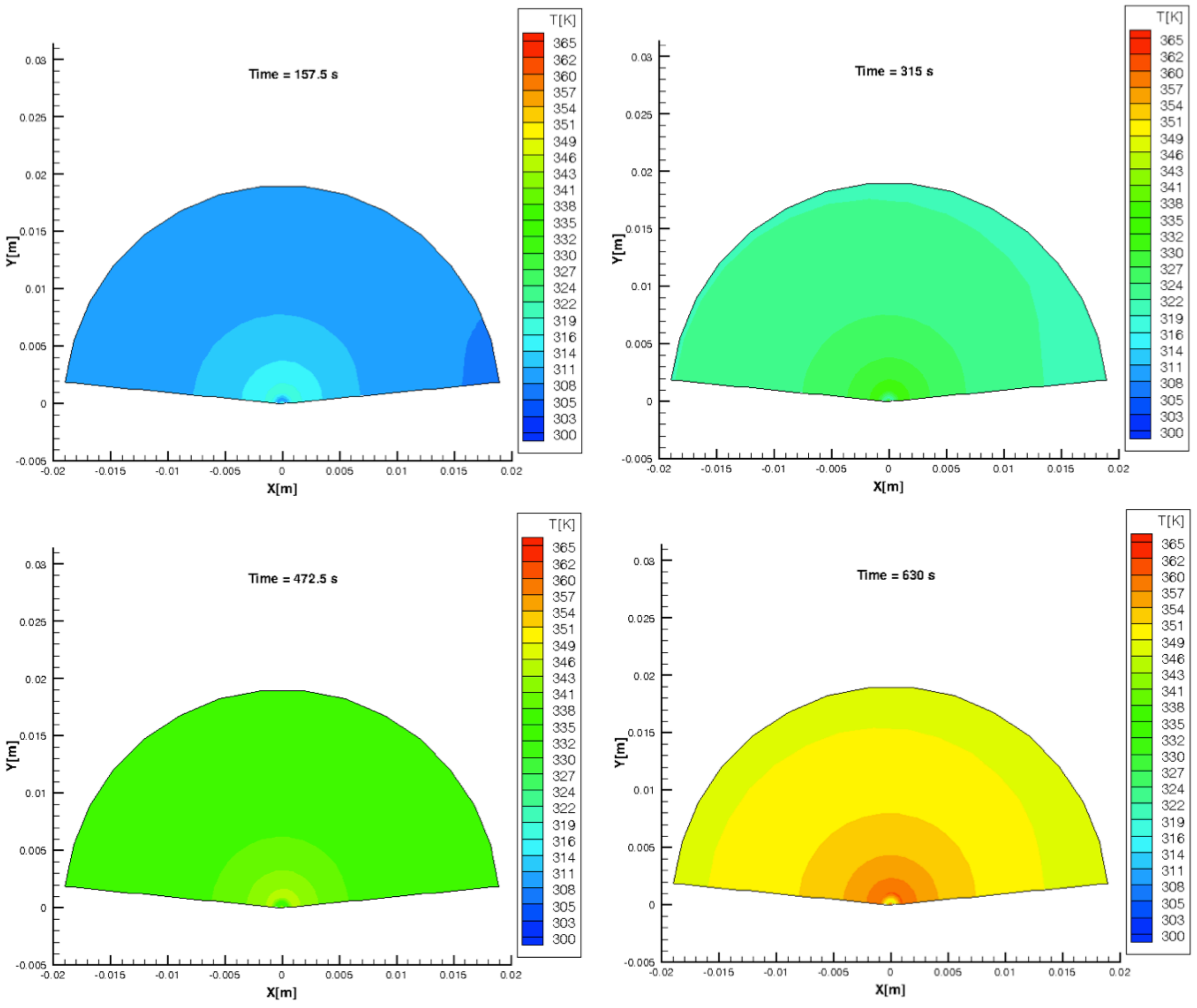


Figure 3: Transient Temperature Distributions for h = 10 W/(m^2*K)

All the temperature field plots presented for both heat transfer coefficient conditions correspond to 30 radial discretizations, 16 angular discretizations, and 1100 time steps. This is degree of spatial discretization is equivalent to 961 cells for an entire circle with 120 cells in the region with a smaller thermal conductivity. These degrees of spatial resolution and temporal resolution were determined to be satisfactory through mesh independence and time step independence analyses. To determine if the coarseness of the mesh significantly affected the results of the simulations, simulations were conducted for both convection coefficients for several radial, angular, and temporal discretizations. The results of the mesh independence studies are indicated in Table 1 and Table 2. The criterion used to determine mesh independence and time step independence is that the relative error of the maximum temperature be less than 2.5% for 4 consecutive discretizations. It is apparent from the tabulated values that according to this criterion, the simulated temperature fields are both mesh independent and time step independent.

Table 1: Results of Mesh Independence Analysis

# of Cells	Relative % Error (h = 2)	Relative % Error (h = 10)
161	**	**
241	1.035574701	1.049899048
321	2.16814538	2.148831843
401	2.0725096	1.883884227
481	1.692122972	1.705492023

Table 2: Results of Time Independence Analysis

# of TimeSteps	Relative % Error (h = 2)	Relative % Error (h = 10)
700	**	**
800	0.293949539	0.283910802
900	0.352141512	0.210084034
1000	0.232851194	0.224376731
1100	0.10708696	0.119381438

Comparison to COMSOLE and Discussion of Error

The results of the COMSOL simulations at the final time step are presented in Figure 4 and Figure 5. Comparison of these results with those generated by the in-house numerical solver indicate that the implemented finite-volume approach accurately predicts the temperature distribution in the cylinder, but overestimates the values of the temperature field. The major source of error between the COMSOL simulations and the results generated by the in-house code is the numerical error due to extremely high connectivity at the center cell/node of the simulated mesh. Since the simulated domain is part of a circular geometry, all the mesh lines must intersect at the central node if polar coordinates are used. Unfortunately, this means that the cell at the center must be connected to "n" number of mesh lines, which must be equal to integral multiples of 8 to guarantee that the boundaries of the cells in the low conductivity region intersect exactly with the thermal conductivity interface.

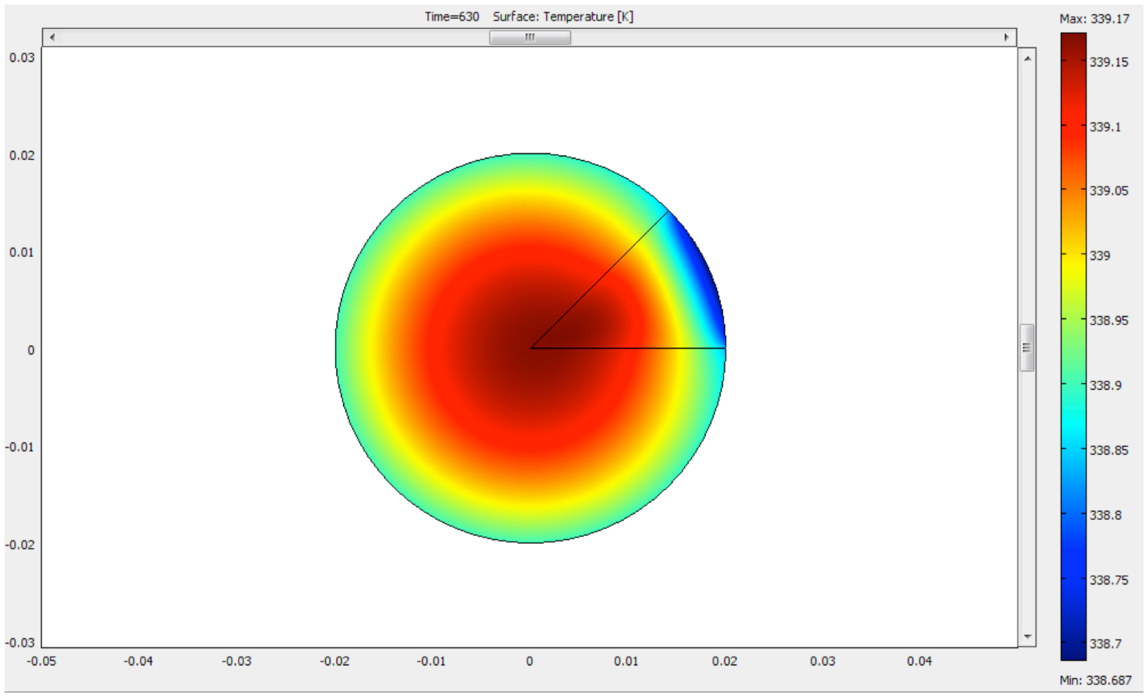


Figure 4: COMSOL Results for h = 2 W/(m^{2}K)

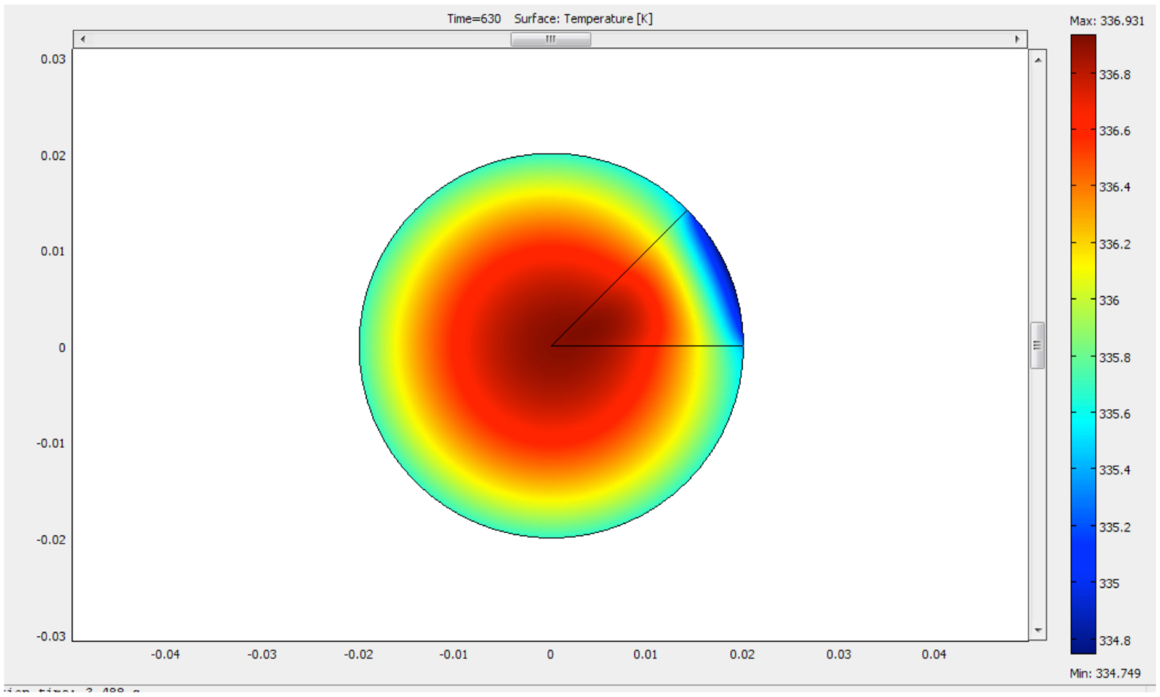


Figure 5: COMSOL Results for h = 10 W/(m^{2}K)

In general, the accuracy and stability of numerical simulations are highly dependent on the structure of the computational domain. A simulation will likely be free of mesh-induced numerical instabilities if all the internal nodes are connected to a fixed number of nodes. This is referred to as "connectivity". For a 2D mesh, a good rule of thumb is that the connectivity should not exceed 6. In the context of heat transfer simulations, a constant, low value for mesh connectivity means that each control volume has to conserve energy based on a small number of surface heat fluxes. If the connectivity of a single node is extremely high, especially compared to the rest of the mesh, the equations conserve the mean energy from several different surface heat fluxes by ascribing a single temperature value to the entire control volume.

The effect of connectivity-associated error on the results of the numerical simulations was evaluated to be a significant source of error because the temperature at the central cell and the interior cells adjacent to the central cell fluctuated drastically. In addition, the temperature field solution was observed to decrease towards the value predicted by the COMSOL simulations as the degree of angular discretization was decreased. This observation is in agreement with the explanation of connectivity-associated error because decreasing the degree of angular discretization decreases the connectivity of the center cell.

APPENDIX

SOLVER CODE: File 1

```
import java.lang.Math;
import java.io.*;

class Solver2 {

	//Domain Parameters
	private double r_max = 0.02;						// cylinder radius - in
meters - [fixed constant]
	private double theta_max = Math.PI;					// angular span of domain
{equal to pi for a half cylinder} - [fixed constant]
	private double theta_secondary_span = Math.PI/8;// angular span of smaller section {angular space
of [0 , theta_secondary_span]} - [fixed constant]

	private double kmajor = 3;						// conductivity of larger
section - in W/(m*K) - [fixed constant]
	private double kminor = 1;						// conductivity of smaller
section - in W/(m*K) - [fixed constant]
	private double To = 298.15;							// initial
temperature of domain - in Kelvin - [fixed constant]
	private double Cp = 950;						// specific heat - in J/
(kg*K) - [fixed constant]
	private double rho = 2618;							// density - in
kg/m^3 - [fixed constant]
	private double Tinf = 298.15;						// far-field temperature -
in Kelvin - specified [fixed constant]

	private double h;								// heat transfer
coefficient - in W/(m^2*K) - [specified constant]

	//Mesh Parameters
	private int r_count;								// # of cells in
the radial direction - [specified constant]
	private int theta_count;						// # of cells in angular
direction - [specified constant]
	private double del_r;								// radial spacing
between cell centers - calculated @ construction [calculated constant]
	private double del_theta;						// angular spacing
between cell centers - calculated @ construction [calculated constant]
	private int cellcount;								// total number
of control volumes - [calculated constant]


	//Simulation Parameters
	private double current_timestep;					//  current timestep
	private double duration = 630;						//	simulation
duration - in seconds - [fixed constant]
```

```
	private int timesteps;										//		# of
timesteps - [specified constant]
	private double del_t;										//
timestep increment - in seconds - [calculated constant]

	//Calculation Variables
	private double LocalCoeff;									// Matrix
Coefficient for center of cell of interest - calculated @ runtime [calculated constant]
	private double EastCoeff;								// Matrix Coefficient for
cell right of cell of interest - calculated @ construction [calculated constant]
	private double WestCoeff;								// Matrix Coefficient for
cell left of cell of interest  - calculated @ construction [calculated constant]
	private double NorthCoeff;								// Matrix Coefficient for
cell above of cell of interest - calculated @ construction [calculated constant]
	private double SouthCoeff;									// Matrix
Coefficient for cell below of cell of interest - calculated @ construction [calculated constant]

	private double K[][];										// Conductivity
Matrix - [calculated constant matrix]
	private double A[][];										// Stiffness
Matrix - [calculated constant matrix]
	private double b[];										//  Variable used
repeatedly to construct the b-matrix - calculated @ runtime for each point of interest

	//Output Variables
	private double[][] T;										//Temperature
field - in Kelvin - Dimensions specified in construction

//T[timestep][mesh_index]

	////////////////////////////////////////
	/////////////Constructor///////////////
	////////////////////////////////////////

	public Solver2(double h_convect, int radial_bands, int angular_bands, int time_steps){
	  //variable assignments
	  h = h_convect;

	  r_count = radial_bands;
	  theta_count = angular_bands;
	  del_r = r_max/(r_count+1);
	  del_theta = theta_max/theta_count;
	  cellcount = r_count*theta_count+1;

	  timesteps = time_steps;
	  del_t = (double)(duration/timesteps);


	  T = new double[cellcount][timesteps];
	  A = new double[cellcount][cellcount];
	  K = new double[r_count][theta_count];
	  b = new double[cellcount];
	}

	///////////////////////////////////////////
	///////////////SOLVER METHODS//////////////
	///////////////////////////////////////////
```

```
        public void ConstructConductivity(){
          double angle;

          for (int theta =0; theta<theta_count; theta++) {
                    angle = theta*del_theta;

                    for (int r = 0; r<r_count; r++) {

                              if(angle<=22.5*Math.PI/180){//minor region
                                        K[r][theta]=kminor;

                              }else{
                                        K[r][theta]=kmajor;//major region
                              }
                    }
          }
        }


        public void AssembleStiffnessMatrix(){
          //central cell
          int i = 0;
          double r,rl,t,rc;

          rc = del_r;
          r = del_r;

          LocalCoeff = Math.PI*rho*Cp/del_t;

          for (int j=1; j<cellcount; j+=r_count) {
                    //rl = r+del_r/2;
                    //t = ((j-r)%theta_count+1)*del_theta;

                    LocalCoeff+=2*(del_theta/rc)*ksouth(j)/(1.5*del_r);


                    A[0][j] = -2*(del_theta/rc)*ksouth(j)/(1.5*del_r);
          }

          A[0][0] = LocalCoeff;


          ////internal cells and boundary cells
          for (i=1; i<cellcount; i++) {
                    r = (i%r_count)*del_r; ///radial position of inner boundary of cell
                    rl = r+del_r/2;//radial position of central point
                    t = ((i-(i%r_count))/r_count+1)*del_theta; //angular position of west cell wall

                    ///boundary cells
                    if(i%r_count==0){
                              LocalCoeff = (rho*Cp/del_t) + keast(i)/((r_max-del_r)*rl*del_theta*del_theta) +
kwest(i)/((r_max-del_r)*rl*del_theta*del_theta)+

                                                                                                   ksouth(i)/
(del_r*del_r)+ h*r_max/((r_max-del_r)*del_r);
                              SouthCoeff = -ksouth(i)/(del_r*del_r);
                              EastCoeff = -keast(i)/((r_max-del_r)*rl*del_theta*del_theta);
                              WestCoeff = -kwest(i)/((r_max-del_r)*rl*del_theta*del_theta);
```

```
                        A[i][i]= LocalCoeff;
                        A[i][i-1]= SouthCoeff;
                        //A[i][i+1]= NorthCoeff(i);

                        if(i>=r_count+1){
                                A[i][i-r_count]= EastCoeff;
                        }

                        if(i<=theta_count*r_count-r_count){
                                A[i][i+r_count]= WestCoeff;
                        }

                        continue;
                }

                //interior cells
                NorthCoeff = -knorth(i)*(r+del_r)/(r*del_r*del_r);
                A[i][i+1]= NorthCoeff;

                if (r>del_r) {//not in contact with center
                        LocalCoeff = (rho*Cp/del_t) + keast(i)/(r*rl*del_theta*del_theta)+kwest(i)/
(r*rl*del_theta*del_theta)+ksouth(i)/(del_r*del_r)+ knorth(i)*(r+del_r)/(r*del_r*del_r);
                        SouthCoeff = -ksouth(i)/(del_r*del_r);
                        A[i][i]= LocalCoeff;
                        A[i][i-1]= SouthCoeff;
                }else{//in contact with center
                        LocalCoeff = (rho*Cp/del_t) + keast(i)/(r*rl*del_theta*del_theta)+kwest(i)/
(r*rl*del_theta*del_theta)+ksouth(i)/(del_r*1.5*del_r)+knorth(i)*(r+del_r)/(r*del_r*del_r);
                        SouthCoeff = -ksouth(i)/(del_r*1.5*del_r);
                        A[i][i]= LocalCoeff;
                        A[i][0]= SouthCoeff;
                }




                if(i>=r_count+1){
                        EastCoeff = -keast(i)/(r*rl*del_theta*del_theta);
                        A[i][i-r_count] = EastCoeff;
                }

                if(i<=theta_count*r_count-r_count){
                        WestCoeff = -kwest(i)/(r*rl*del_theta*del_theta);
                        A[i][i+r_count] = WestCoeff;
                }
          }
        }

        public void Initialize(){

          for(int i =0;i<cellcount;i++){
                T[i][0]=To;
          }

        }

        public void updateB(double delt,int next_timestep){
```

```
        b[0]= T[0][next_timestep-1]*Math.PI*rho*Cp/del_t + qgen(next_timestep*delt)*Math.PI;

        for (int i = 1; i<cellcount; i++) {
                if(i%r_count==0)
                {//boundary cell
                        b[i]=T[i][next_timestep-1]*rho*Cp/delt + qgen(next_timestep*delt)+h*Tinf*r_max/
((r_max-del_r)*del_r);
                        continue;
                }

                //interior cells
                b[i]=T[i][next_timestep-1]*rho*Cp/del_t+qgen(next_timestep*delt);

        }
    }


    public void Solve(){
        double[] temp;

        for (int i=1; i<timesteps; i++) {
                System.out.println( i);
                updateB(del_t, i);
                temp=SOR(A,b,0.05,To,1);//forces convergence to within 0.1%

                for (int j =0; j<cellcount; j++){//copy solution to solution matrix
                        T[j][i]=temp[j];
                }

        }

    }

    public double[] getT(int time_step){
        double[] out =    new double[cellcount];

        for (int j =0; j<cellcount; j++){//copy solution to solution matrix
                out[j]=T[j][time_step];
        }

        return out;

    }

    public double keast(int i){
        //i is greater or equal to 1
        int r1,t1,r2,t2;
        r1 = (i-1)%r_count;
        t1 = (i-r1)/r_count;
        r2 = r1;
        t2 = t1-1;

        if (t1>0) {
                return 2*K[r1][t1]*K[r2][t2]/(K[r1][t1]+K[r2][t2]);
        }else{
                return 0.0;
        }
```

```
}

public double kwest(int i){
  //i is greater or equal to 1
  int r1,t1,r2,t2;
  r1 = (i-1)%r_count;
  t1 = (i-r1)/r_count;
  r2 = r1;
  t2 = t1+1;

  if (t1<theta_count-1) {
          return 2*K[r1][t1]*K[r2][t2]/(K[r1][t1]+K[r2][t2]);

  }else{
          return 0.0;
  }
}

public double ksouth(int i){
  //i is greater or equal to 1
  int r1,t1;
  r1 = (i-1)%r_count;
  t1 = (i-r1)/r_count;
  return K[r1][t1];
}

public double knorth(int i){//for internal elements
  return ksouth(i);
}

public void writetoFile(int t_step, double h, int radd, int thetadd){
  double[][] out = new double[cellcount][3];

  for (int i = 0; i<cellcount; i++) {
          int r,t;
          double rr, tt;
          double x,y;

          r = i%r_count;
          t = (i-r)/r_count;

          rr=del_r/2+r*del_r;
          tt=(t+0.5)*del_theta;

          x=rr*Math.cos((t+0.5)*del_theta);
          y=rr*Math.sin((t+0.5)*del_theta);

          if(i==0){
                  x=0.0;
                  y=0.0;
          }


          out[i][0]=x;//x-coordinate
          out[i][1]=y;//y-coordinate
          out[i][2]=T[i][t_step];//temperature
  }
```

```
          Writer pen = new Writer();
          String filename = new String("T-Cartesian, Time_Step = " + t_step + ", h = " + h + ", radd = "
+radd+ ", thetadd = " + thetadd);
          pen.writeVector(out,filename);

        }

        ////-----Internal Heat Generation Function------////////
        public double qgen(double time){
          double c1,c2,c3,c4,c5,c6,c7;
          c1 = -1.7031*Math.pow(10,-11);
          c2 =  1.3146*Math.pow(10,-7);
          c3 = -1.8011*Math.pow(10,-4);
          c4 = 1.024*Math.pow(10,-1);
          c5 = -2.8133*Math.pow(10,1);
          c6 = 3.6444*Math.pow(10,3);
          c7 = -2.7545*Math.pow(10,3);

          double out = c1*Math.pow(time,6) + c2*Math.pow(time,5) + c3*Math.pow(time,4) +
                                  c4*Math.pow(time,3) + c5*Math.pow(time,2) + c6*Math.pow(time,1)+ c7;

          return out;

        }

        ///Test Function
        public double[][] getK(){
          double[][] out =new double[r_count][theta_count];

          for (int theta = 1; theta<theta_count+1; theta++) {
                  for (int r = 1; r<r_count+1; r++) {
                          out[r-1][theta-1]=K[r-1][theta-1];
                  }
          }

          return out;

        }

        //////----Successive Over-Relaxation Matrix Solver----/////////
        public double[] SOR(double[][] AA, double[] bb, double tol, double guess, double w){
          //////Uses the Successive Over Relaxation Method to Solve the system Ax=b
          //w>1 accelerates convergence but risks divergence
          //w<1 forces an unstable system to converge

          int n = AA.length;
          double[] xk_1 = new double[n];
          double[] xk = new double[n];

          double sigma = 0.0;
          double margin = 0.0;
          double[] temp;


          //Initializes output vector with initial guess
          for (int i=0; i<n; i++) {
                  xk_1[i]=guess;
```

```
                xk[i]=guess*2;
        }

        margin = 100;//arbitrarily large initial percentile error

        while (margin>tol) {//checks convergence

                for (int i =0; i<n; i++) {
                        xk_1[i]=xk[i]; //assigns kth-iteration of x to xk_1 before next iteration
                }

                for (int i=0; i<n; i++) {

                        sigma = 0;

                        for (int j=0; j<n; j++) {

                                if (j!=i){sigma = sigma + AA[i][j]*xk[j];}

                        }//end of j-loop

                        xk[i]=(1-w)*xk[i]+(w/AA[i][i])*(bb[i]-sigma);//updates xk with new values

                }//end of i-loop: kth iteration has completed

                margin = maxerror(xk_1,xk);

        }//end of while loop

        return xk;
      }

      ///////----Maximum percentile difference between elements----//////////
      private double maxerror(double[] x1, double[] x2){
        double out=0;
        double diff = 0;

        for(int i=0; i<x1.length; i++){
                diff=100*Math.abs(((x1[i]-x2[i])/x1[i]));

                if(diff>out){out=diff;}//replaces lower error with greater error

        }
        return out;//returns greatest error
      }


}
```

SOLVER CODE: File 2

```
import java.io.*;
import java.lang.Math;

class Main{
```

```
public static void main(String[] args){
   //int radd=10; //-- 641 cells for circle
   //int radd=15; //-- 961
   //int radd=20; //-- 1281
   //int radd=25; //-- 1501
   int radd = 30;

   //int radd = 50;
   //int thetad = 16;

   int thetad=16;
   double h = 2;

   //int t_steps = 700; //0.9 seconds
   //int t_steps = 800; //0.78 seconds
   //int t_steps = 900; //0.7 seconds
   //int t_steps = 1000; //0.63 seconds
   int t_steps = 1100; //0.63 seconds

   //for(int radd = 10; radd<26; radd +=5){

   //thetad MUST be integral multiples of 8

   Solver2 solver = new Solver2(h,radd,thetad,t_steps);
   //Solver2 solver = new Solver2(10,radd,thetad,400);
   //double[][] A = new double[4][4];
   //double[] b = new double[4];
   double[] out;

   solver.ConstructConductivity();
   solver.AssembleStiffnessMatrix();
   solver.Initialize();
   solver.Solve();

   //out=solver.getT(t_steps-1);
   //out= solver.getK();

   //System.out.println(       "Solution");
   //for (int i = 0; i<radd*thetad; i++) {
   //           System.out.println(         out[i]);
   //}

   solver.writetoFile(275,h,radd,thetad);
   solver.writetoFile(550,h,radd,thetad);
   solver.writetoFile(825,h,radd,thetad);
   solver.writetoFile(t_steps-1,h,radd,thetad);
   /*for (int theta =0; theta<thetad; theta++) {

               for (int r = 0; r<radd; r++) {
                           System.out.println(out[r][theta]);

               }
   }*/
   //}
}
```

```
}
```

SOLVER CODE: File 3

```
/** Simple Program to write a text file
 */

import java.io.*;

public class Writer{

        public Writer(){}

        public void writeVector(double[][] input, String file){
           try {
                FileWriter outFile = new FileWriter(file);
                PrintWriter out = new PrintWriter(outFile);

                for(int i = 0; i<input.length; i++){
                        out.println(input[i][0] + " " + input[i][1] + " " + input[i][2]);
                }

                out.close();
        } catch (IOException e){
                e.printStackTrace();
        }
      }
}
```